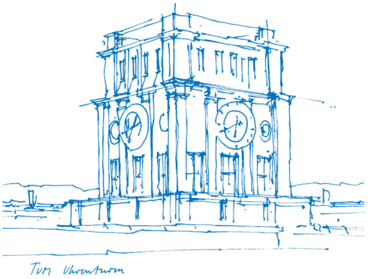
CUDA Introduction

HPC - Algorithms and Applications

Alexander Pöppl
Technical University of Munich
Chair of Scientific Computing

October 31st 2018



References

- D. Kirk, W. Hwu:
  Programming Massively Parallel Processors, Morgan Kaufmann, 2010

Part I

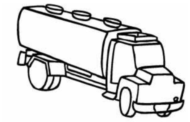
Introduction to CUDA

Motivation

Currently there are two opportunities of parallelizing programs

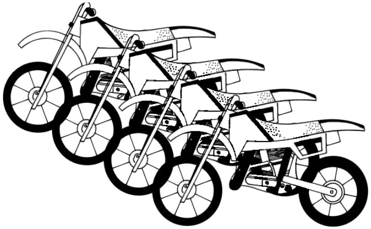
multi-cores

- distribute the work on few strong multi-purpose processors (CPUs)
- regular supercomputers, clusters
- OpenMP, MPI



many-cores

- distribute the work on a lot of single purpose processors
- Intel MIC, BlueGene, GPUs



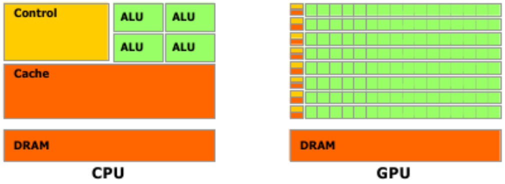
Difference CPU / GPU

CPU

- general purpose
- large amount of transistors for non-computational tasks
- allows out of order execution
- pipelining
- optimized for sequential tasks

GPU

- many processors dedicated to computations
- less support for branching
- well aligned data streamed through processor – data parallelism



GPU Computing – Timeline

- 80s/90s: fixed-function graphics pipelines
- 2000: programmable real-time graphics, vertex/pixel shaders
- 2003: ”GPGPU”, parallel programmable hardware exploited for general purposes
- 2007: GPU Computing, many-core architecture, CUDA and OpenCL programming models

Different Programming Models for GPU

CUDA

- GPU - only
- standard formed by vendor (nVidia)
- adopts new architectures fast

OpenCL

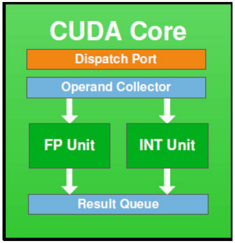
- standard formed by consortium (Khronos Group)
- platform independent (also for ATI and CPU's)
- slower development

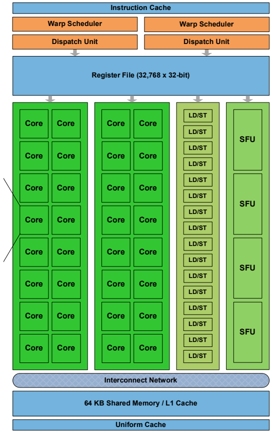
We will use CUDA

- interface is easier to learn
- paradigms of GPU programming better understandable

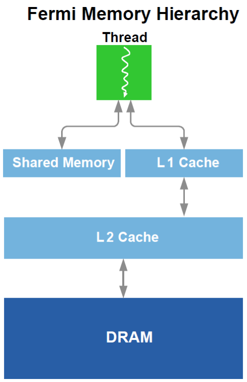
GPU Architectures

NVIDIA Fermi





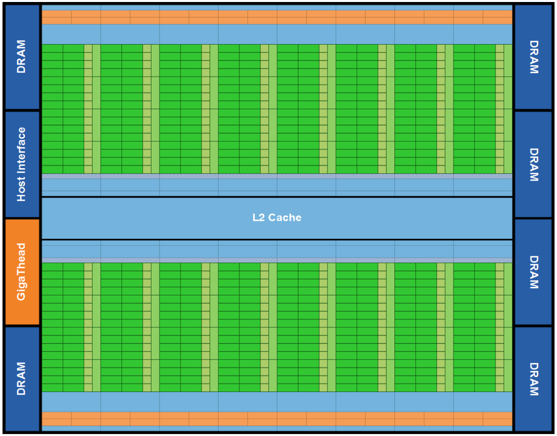
Fermi Streaming Multiprocessor (SM)



(source: NVIDIA – Fermi Whitepaper)

GPU Architectures

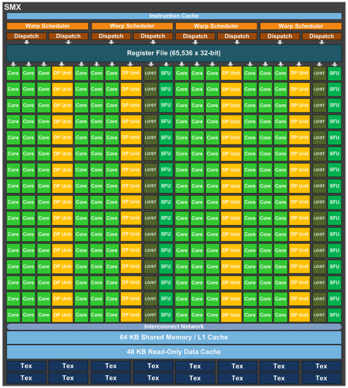
NVIDIA Fermi Microarchitecture

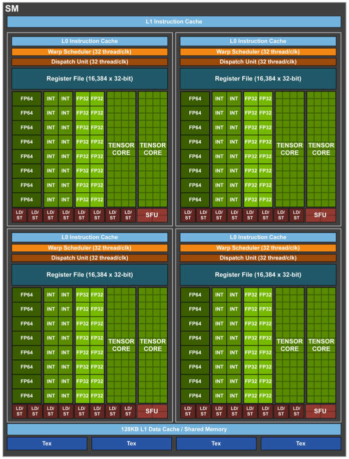


(source: NVIDIA – Fermi Whitepaper)

GPU Architectures

NVIDIA Kepler/Volta Streaming Multiprocessors

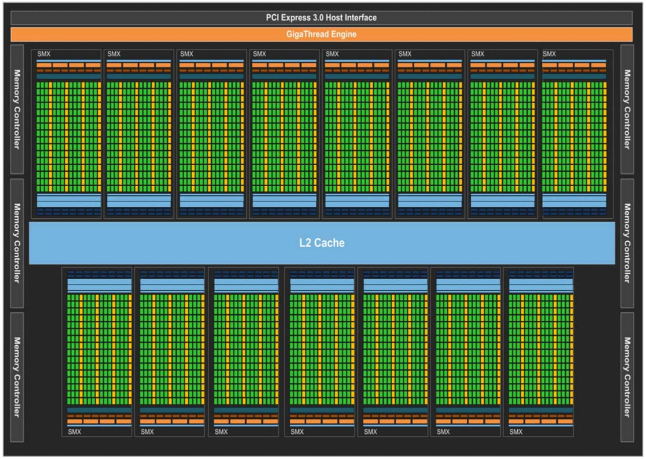

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

(sources: https://devblogs.nvidia.com/parallelforall/inside-volta/ & Kepler Whitepaper)

GPU Architectures

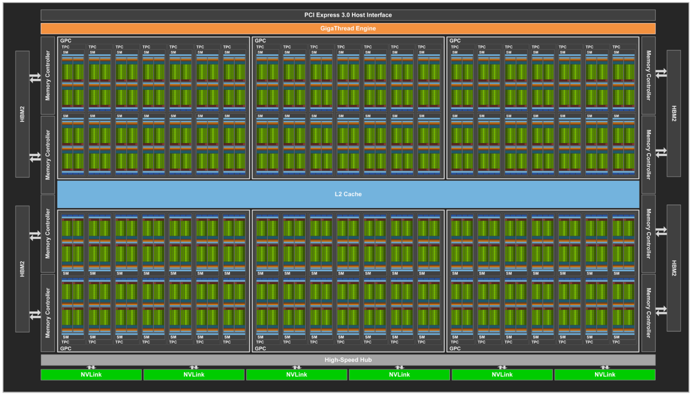
NVIDIA Kepler Microarchitecture



(source: NVIDIA – Kepler Whitepaper)

GPU Architectures

NVIDIA Volta Microarchitecture



(source: https://devblogs.nvidia.com/parallelforall/inside-volta/)

Part II

CUDA Basics

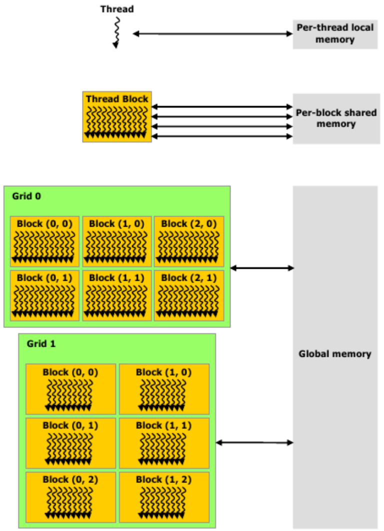
CUDA – Architecture Model

Host & Device:

- host = CPU with main memory
- device(s) = GPU/coprocessor(s) with device memory

Hardware characteristics:

- massively parallel (thousands of cores)
- lightweight threads, hardware-supported
- massive parallelism hides memory latency;
  focus on data parallelism



Warps

- warp: set of 32 concurrent threads in a block
- only one (Fermi) / two (Kepler) instruction(s) per cycle per warp

Question: what happens when a warp executes a branch?

Host Memory

- slow access by PCI bus
- used as little as possible

CUDA – Programming Model

CUDA as extension of C:

- host code (program control) and device code (GPU) combined in a single C program
- device code consists of massively parallel *kernels* that are off-loaded to the GPU
- language extension for defining and calling kernels
- API function to allocate device/host memory, synchronize threads, etc.
- SIMD/SPMD (single instruction/program, multiple data)

Example: Matrix Multiplication

General Approach: PRAM program

```
for i from 1 to n do (in parallel?)
  for k from 1 to n do (in parallel?)
    for j from 1 to n do (in parallel?)
      C[i,k] += A[i,j]*B[j,k]
```

Example: Matrix Multiplication

General Approach: PRAM program

```
for i from 1 to n do in parallel
  for k from 1 to n do in parallel
    for j from 1 to n do
      C[i,k] += A[i,j]*B[j,k]
```

- PRAM: executed on n^2 processors
- CUDA: n^2 CUDA threads; each thread executes one j-loop (i.e., computes one element `C[i,k]`)
- part 1: memory transfer (host$\rightarrow$device and device$\rightarrow$host)
- part 2: launch/execution of kernel code for j-loop

CUDA Memory transfer

Memory instructions in CUDA:

- Memory allocation:
```
cudaMalloc(void** ppd, int size);
```
- Memory deallocation:
```
cudaFree(void* pd);
```
- Copy from host to device:
```
cudaMemcpy(pd, p, size, cudaMemcpyHostToDevice);
```
- Copy from device to host:
```
cudaMemcpy(p, pd, size, cudaMemcpyDeviceToHost);
```

Matrix Multiplication – Memory Transfer

```
__host__ void matrixMult(float *A, float *B,
float *C, int n);
```

Task:

- Write the memory transfer code for a matrix multiplication $C \leftarrow C + A \cdot B$, i.e. allocate device data, copy data between host and device and deallocate device data. All matrices have size $n \times n$.

Matrix Multiplication – Memory Transfer

```
__host__ void matrixMult(float *A, float *B,
float *C, int n) {
    int size = n*n*sizeof(float);
    float* Ad; float* Bd; float* Cd;
    cudaMalloc((void**)&Ad, size);
    cudaMalloc((void**)&Bd, size);
    cudaMalloc((void**)&Cd, size);
    cudaMemcpy(Ad,A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd,B, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Cd,C, size, cudaMemcpyHostToDevice);
    /* ... perform multiplication on device ... */
    cudaMemcpy(C,Cd, size, cudaMemcpyDeviceToHost);
    cudaFree(Ad); cudaFree(Bd); cudaFree(Cd);
}
```

Grids and Blocks in CUDA

Blocks:

- threads can be organized as 1D, e.g. (128,1,1), 2D, e.g. (16,16,1), or 3D, e.g. (4,8,16) blocks
- limited to 1024 (Fermi, Kepler), 512 (GT2xx) threads per block
- threads in one block are always executed in parallel
- and can use separate, shared memory

Grids:

- CUDAcc $< 2.x$: 2D layout. Today: 3D layout allowed
- up to $2^{32} \times 2^{32}$ ($\geq$Kepler) / $2^{16} \times 2^{16}$ ($\leq$ Fermi) blocks per grid
- blocks in a grid may be executed in parallel
  (but, in practice, will be scheduled to available cores)

Kernel Invocation: Grids and Blocks

```
/* ... */
dim3 dimBlock(n,n);
dim3 dimGrid(1,1);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */
```

- threads are combined to 3D **blocks**:
  $\rightarrow$ `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
- blocks are combined to 2D **grids**:
  $\rightarrow$ `blockIdx.x`, `blockIdx.y`

Matrix Multiplication – CUDA Kernel

```
__global__ void matrixMultKernel(float* Ad, float* Bd,
float* Cd, int n) {
    int i = threadIdx.x;
    int k = threadIdx.y;
    float Celem = 0;
    for(int j=0; j<n; j++) {
        float Aelem = Ad[i*n+j];
        float Belem = Bd[j*n+k];
        Celem += Aelem*Belem;
    }
    Cd[i*n+k] += Celem;
}
```

Matrix Multiplication – with Grid

```
__global__ void matrixMultKernel(float* Ad, float* Bd,
float* Cd, int n);
```

Task:

- Extend the kernel from a single block to a grid with multiple blocks. Each block should have the size `TILE_SIZE` × `TILE_SIZE`.
- Change grid and block dimensions in the kernel invocation accordingly

Matrix Multiplication – with Grid

```
__global__ void matrixMultKernel(float* Ad, float* Bd,
float* Cd, int n) {
    int i = blockIdx.x * TILE_SIZE + threadIdx.x;
    int k = blockIdx.y * TILE_SIZE + threadIdx.y;
    float Celem = 0;
    for(int j=0; j<n; j++) {
        float Aelem = Ad[i*n+j];
        float Belem = Bd[j*n+k];
        Celem += Aelem*Belem;
    }
    Cd[i*n+k] += Celem;
}
```

Matrix Multiplication – with Grid (2)

```
/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */
```

- What is the optimal tile size?

Matrix Multiplication – with Grid (2)

```
/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */
```

- What is the optimal tile size?
- Too small $\rightarrow$ large overhead, low performance
- Too large $\rightarrow$ block size limit, register limit
- `TILE_SIZE^2` $\leq$ max. threads per block
  Fermi, Kepler: 1024 $\Rightarrow$ choose `TILE_SIZE = 32` (for now)
- If $n \mod 32 \neq 0$: padding of matrix to match tile size
  $(n \leftarrow \lceil \frac{n}{32} \rceil \cdot 32)$

Matrix Multiplication – with Grid (2)

```
/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */
```

- What is the optimal tile size?
- Too small $\rightarrow$ large overhead, low performance
- Too large $\rightarrow$ block size limit, register limit
- `TILE_SIZE^2` $\leq$ max. threads per block
  Fermi, Kepler: 1024 $\Rightarrow$ choose `TILE_SIZE = 32` (for now)
- If $n \mod 32 \neq 0$: padding of matrix to match tile size
  ($n \leftarrow \lceil \frac{n}{32} \rceil \cdot 32$)

Let's compare CPU/GPU performance!

Matrix Multiplication – with Grid (2)

```
/* ... */
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(n/TILE_SIZE,n/TILE_SIZE);
matrixMultKernel<<<dimGrid,dimBlock>>>(Ad,Bd,Cd,n);
/* ... */
```

- What is the optimal tile size?
- Too small $\rightarrow$ large overhead, low performance
- Too large $\rightarrow$ block size limit, register limit
- `TILE_SIZE^2` $\leq$ max. threads per block
  Fermi, Kepler: 1024 $\Rightarrow$ choose `TILE_SIZE = 32` (for now)
- If $n \mod 32 \neq 0$: padding of matrix to match tile size
  $(n \leftarrow \lceil \frac{n}{32} \rceil \cdot 32)$

Let's compare CPU/GPU performance! $\Rightarrow$ Okay, that sucks.

CUDA Memory

Types of **device** memory in CUDA:

- per thread: **registers** and **local memory**
  (locally declared variables and arrays (local memory),
  $\rightarrow$ lifetime: kernel execution)
- per block: **shared memory**
  (keyword `__shared__`, lifetime: kernel execution)
- per grid: **global memory** and **constant memory**
  (keywords `__device__`, `__constant__`;
  lifetime: entire application)
- vs.: CPU cache hierarchy (registers, caches, RAM)

Matrix Multiplication – Performance Estimate

Multiplication kernel:

```
for(int j=0; j<n; j++) {
    float Aelem = Ad[i*n+j];
    float Belem = Bd[j*n+k];
    Celem += Aelem*Belem;
}
```

- NVidia GT-650M stats: memory bandwidth: 64 GB/s, floating point performance: 691 GFlop/s
- two floating-point operations (multiply and add) per two floating-point variables (each 4 byte)
- thus: max. of 16 Giga `float` variables can be transferred from global memory per second
- limits performance to $<$ 16 GFlop/s

Matrix Multiplication with Tiling

- observation: simple matrix multiplication kernel is slow (far below peak performance)
- anticipated reason: redundant access to slow global memory; performance limited by memory bandwidth between global memory and CUDA cores (each entry is loaded n times)

Remedy: **Tiling**

- switch to tile-oriented implementation (matrix multiplication on sub-blocks)
- copy matrix tiles into shared memory
- let all threads of a block work together on shared tile
- accumulate result tile back on matrix in global memory

A Note on Synchronization

Barrier-synchronization in CUDA:

```
__syncthreads();
```

- barrier for all threads within a block
- usual rules: all threads need to execute (or not) the same(!) call to `__syncthreads()`
- threads of the same block scheduled to the same hardware unit
- in contrast: no synchronization features for threads in a grid $\rightarrow$ reason: *transparent scheduling* of entire blocks

Matrix Multiplication – with Tiles

```
__global__ void matrixMultKernel(float* Ad, float* Bd,
float* Cd, int n) {
    __shared__ float Ads[TILE_SIZE][TILE_SIZE];
    __shared__ float Bds[TILE_SIZE][TILE_SIZE];
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int i = blockIdx.x * TILE_SIZE + tx;
    int k = blockIdx.y * TILE_SIZE + ty;
    /* (cont.) */
```

Task:

- Copy matrix tiles of A and B from global to shared memory.
- Execute matrix multiplication on shared tiles and write the result to matrix C in global memory.
- Synchronize the thread block where it is necessary.